Spring 5-1-1999

Exploring Last n Value Prediction ; CU-CS-885-99

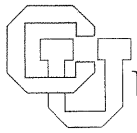
Martin Burtscher
University of Colorado Boulder

Benjamin G. Zorn
Microsoft Corporation

Exploring Last n Value Prediction

Martin Burtscher
Benjamin G. Zorn

CU-CS-885-99



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

Exploring Last *n* Value Prediction

Technical Report CU-CS-885-99

Martin Burtscher

Department of Computer Science
University of Colorado
Boulder, Colorado 80309-0430
burtsche@cs.colorado.edu

Benjamin G. Zorn

Microsoft Corporation
1 Microsoft Way
Redmond, WA 98052
zorn@microsoft.com

Abstract

Load instructions occasionally incur very long latencies that can significantly affect system performance. Load value predictors alleviate this problem by enabling the CPU to speculatively continue processing without having to wait for the slow memory access to complete.

In this paper we explore a technique to improve both the accuracy and in particular the coverage of a basic last value predictor by increasing its width to retain the last *n* values. For example, a modest 16kB predictor running SPECint95 turns out to perform best when retaining the last four values.

Detailed pipeline-level, cycle-accurate simulations of a superscalar microprocessor with various load value predictors show that our last four value predictor outperforms other predictors from the literature, often significantly.

The 16kB predictor yields a harmonic mean speedup of 13.7% with a not yet realized re-execute misprediction recovery mechanism and 12.5% with existing re-fetch recovery hardware. The re-fetch speedup may be close enough to the re-execute speedup to render the more complex re-execution mechanism unnecessary.

1. Introduction

Due to their occasional long latency, load instructions can have a significant impact on system performance. If the gap between CPU and memory speed continues to widen, this latency will become even more detrimental. Since loads are not only among the slowest but also among the most frequently executed instructions of current high-performance microprocessors [LCB+98], improving their execution speed should significantly improve the overall performance of the processor.

Fortunately, load values are quite predictable. For instance, about half of the executed load instructions of the SPECint95 benchmark suite retrieve the same value that they did the previous time they were executed. This behavior, which has been observed explicitly on a number of architectures, is referred to as *value locality* [LWS96, Gab96].

Empirically, papers have shown that the results of most instructions are predictable [Gab96, LiSh96, SaSm97a]. However, of all the frequently occurring, result-generating instructions, load instructions are the most predictable [LiSh96] and incur the longest latencies. Since about every fifth executed instruction is a load, predicting only load values requires significantly fewer predictions and leaves more time to update the predictor. As a consequence, smaller and simpler predictors can be used. We therefore believe that predicting only load values may well be more cost effective than predicting the result of every instruction.

Load value predictors try to exploit the existing value locality. To reduce the number of mispredictions, the predictors usually contain both a *value predictor* and a *confidence estimator* (CE) to decide whether or not to make a prediction. The CE only allows predictions to take place if the confidence that the prediction will be correct is high. This is essential because sometimes the value predictor does not contain the necessary information to make a correct prediction. In such a case, it is better not to make a prediction because incorrect predictions incur a cycle penalty (for undoing the speculation) whereas making no prediction does not.

CEs are similar to branch predictors in the sense that both make binary decisions (predictable or not-predictable and branch taken or not-taken, respectively). Therefore, we adopt the nomenclature from the branch prediction literature to describe the CEs.

Most of the proposed load value predictors include *bimodal* [McF93] confidence estimators, i.e., they use saturating counters to "count" how frequently predictions turned out to be correct in the recent past. If this count is above a given threshold, meaning that there were at least a certain number of potentially correct predictions in the past, the predictor is allowed to make further predictions. Otherwise, predictions are temporarily inhibited until the count becomes high enough again.

In previous work [BuZo98, BuZo99] we adopted a different idea from the branch prediction literature to build a more accurate CE and hence a more accurate predictor. Our CE is in essence an *SAg* predictor [YePa93] that keeps a small history recording the most recent prediction outcomes (success or failure) [SCAP97]. The different history patterns are then

used to decide whether to make a value prediction or not. In this paper we use that same CE but we now explore possibilities to improve the predictor's coverage, that is, to increase the number of prediction attempts without loss of accuracy.

If load values are predicted quickly and correctly, the CPU can start processing the dependent instructions without having to wait for the memory access to complete, which potentially results in a significant performance increase. Of course, it is only known whether a prediction was correct once the true value has been retrieved from the memory, which can take many cycles. *Speculative execution* allows the CPU to continue execution with a predicted value before the prediction outcome is known [SmSo95]. Because branch prediction requires a similar mechanism, most modern microprocessors already contain the required hardware to perform this kind of speculation [Gab96].

Unfortunately, branch misprediction recovery hardware causes all the instructions that follow a misspeculated instruction to be purged and *re-fetched*. This operation is costly and makes a high prediction accuracy paramount. Unlike branches, which invalidate the entire execution path when mispredicted, mispredicted loads only invalidate the instructions that depend on the loaded value. In fact, even the dependent instructions per se are correct, they just need to be *re-executed* with the correct input value(s). Consequently, a better recovery mechanism for load misspeculation would only re-execute the instructions that depend on the mispredicted load value. Such a recovery policy is less susceptible to mispredictions and favors a higher coverage, but may be prohibitively hard to implement.

Our load value predictor's re-fetch performance is not only high but also close to its re-execute performance, making the added benefit of a re-execution core small in comparison. Perhaps this is an indication that complex re-execution hardware may not be needed.

Load value predictors normally consist of an array of slots to store information about recently executed loads. It is this information that is used for making a prediction the next time a load is executed. Once a predictor contains enough slots to hold information about all the frequently executed loads, increasing the number of slots does not improve the predictor's performance significantly because the additional slots will at best be used for predicting infrequently and therefore unimportant load instructions. As an alternative, we suggest using extra real-estate to increase the amount of information in each slot instead of the number of slots. This choice should enable the predictor to make better and/or more predictions and thus improve its performance.

Running SPECint95, quite small predictors already benefit more from storing additional information in the slots than from increasing the number of slots. For example, our 16kB predictor with 512 slots each holding the last four loaded values performs better than the same predictor with 1024 slots holding the last two values or with 256 slots holding the last eight values. Section 5.2.2 provides more detail.

Our last four value predictor also outperforms other predictors from the literature and reaches an average speedup over SPECint95 of 13.7% with a re-execute misprediction recovery policy and 12.5% with a re-fetch recovery policy. Section 5.1.2 provides more detailed results.

The remainder of this paper is organized as follows: Section 2 introduces the architecture of our last four value predictor. Section 3 presents related work. Section 4 explains the methods used. Section 5 presents the results. Section 6 concludes the paper with a summary.

2. The SAg Last *n* Value Predictor

Figure 2.1 shows the architecture of our last four value predictor. The predictor is composed of four identical components, each of which consists of an array of 512 slots for storing a 64-bit value and a ten-bit prediction outcome history. Furthermore, each component has an array of 1024 four or five-bit saturating up/down counters associated with it. The prediction outcome histories together with the saturating counters represent the SAg confidence estimator. Each of the 512 lines of the predictor contains an eight-bit partial tag, which is shared between the four components. In this configuration, our predictor requires 21kB of state.

Predictions are made in the following way: First, the nine least significant bits from the load instruction's program counter (PC) value are used to index one of the predictor's lines (direct mapping). If the partial tag of that line does not match, no prediction is made. Otherwise, the four components each predict (in parallel) the value stored in the selected slot. At the same time, the components use the selected history to index a counter in their array of saturating counters. The four resulting counter values are then compared with each other and whichever component happens to report the highest counter value is selected to make the actual load value prediction if its counter value is also above a preset threshold.

If multiple components report the same maximum confidence (and that confidence is not below the preset threshold), one of the components has to be chosen over the others. We tried two prioritizing schemes: giving the component with the youngest value priority and giving the component with the oldest value priority. There is almost no difference between the two approaches, but the former always seems to outperform the latter. We will therefore prioritize from young to old.

Once the outcome of a prediction is known, the predictor needs to be updated. This process is similar to making a pre-

diction except that each component compares its predicted values with the true load value instead of making a prediction. If the two values are identical, the selected counter is incremented (unless it has already reached its maximum) and a one (indicating a success) is shifted into the selected prediction outcome history. If the two values differ, the corresponding counter is decremented by a preset penalty (but not below zero) and a zero (indicating a failure) is shifted into the prediction outcome history. Finally, the four values in the selected line are shifted over to the next component, i.e., the oldest value is lost, component four gets the value from component three etc. and component one receives the just loaded value. Thus, the first component always contains the most recent load value, the second component the second most recent value and so on.

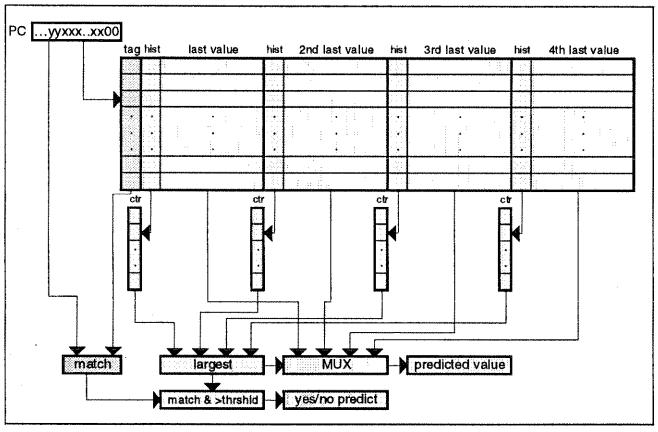


Figure 2.1: The architecture of our tagged SAg last four value predictor.

3. Related Work

In this section we present related work and we introduce all the predictors that we later compare our predictor against. To make the comparison as fair as possible, we scale all predictors to a size of 16kB for retaining load values plus whatever else they need to support this size, since we believe 16kB to be a reasonable predictor size. We mention the total predictor sizes to give an idea of the relative complexity. Note that the indicated size is not generally the size originally used by the authors of the individual predictors. We therefore also show simulation results for other predictor sizes (4kB - 64kB) in Section 5.1.2.

The predictor most closely related to our own is Wang and Franklin's last distinct four value predictor (*LD4V*) [WaFr97]. Their predictor retains only distinct values (ours does not) and uses a least recently used replacement policy. Instead of prediction outcome histories, their predictor keeps a history of which slots were most recently accessed. This history is used to index four arrays of saturating counters that correspond to the four components. The maximum counter value determines which component is selected to make a prediction. Predictions only take place if the counter value is above a preset threshold. An *LD4V* storing 16kB of load values requires a total of about 26kB of state.

To improve the performance of the *LD4V* predictor, Wang and Franklin propose hybridizing an *LD4V* and a stride predictor [WaFr97]. We call the resulting predictor *LD4V Stride*. A stride predictor predicts a value that is the sum of the last value plus an offset (stride). This offset is computed by taking the difference between the last value and the second to last value. Since the stride component only stores 8-bit partial strides and uses the values from the *LD4V* component, the hybrid predictor is not significantly larger than *LD4V* and requires 27kB of state. Wang and Franklin did not perform cycle-level performance simulations of either of their predictors.

In previous work we have developed a tagged SAg-based last value predictor (*Tag SAg LV*) [BuZo98], which is identical to the last four value predictor presented in this paper except it only contains one component instead of four. In spite of this similarity, *Tag SAg L4V* significantly outperforms *Tag SAg LV*.

The predictor most closely related to our *Tag SAg LV* is Lipasti and Shen's bimodal last value predictor (*Bim LV*) [LiSh96]. It is untagged and uses two-bit saturating counters as a confidence estimator. The counters are located where our predictor keeps the histories. A value prediction only takes place if the corresponding counter value is above a given threshold. *Bim LV* is the smallest predictor with a size of 17kB.

We expand on *Bim LV* by adding partial tags and performing a detailed parameter space analysis to find the optimal CE setting. As it turns out, three-bit counters with a penalty above one perform the best. Because of the somewhat larger counters and the extra tags, the size of this predictor (*Tag Bim LV*) amounts to 19kB.

We performed another detailed analysis for the *Tag Bim St2d* predictor, which is a tagged stride 2-delta [SaSm97a] predictor with a bimodal CE. A 2-delta stride predictor retains two strides. The stride used to compute the next prediction is only updated if a new stride has been seen at least twice in a row. This results in significantly better performance than a conventional stride predictor can deliver. The size of the *Tag Bim St2d* is 23kB due to the two partial stride fields.

The last predictor we compare against is *St2d FCM*, which is similar to the one presented by Rychlik et al. [RFKS98] except it is not set-associative. The predictor is a

hybrid between a stride 2-delta and a finite context method-based (FCM) predictor [SaSm97b]. FCM predictors store entire sequences of load values. Upon prediction they try to identify the current location in the sequence and use the next value from the sequence to make a prediction. The configuration that yields the best result requires about 26kB of state with re-execute and 29kB of state with re-fetch, which makes it the largest predictor.

4. Methodology

All our measurements are performed on the DEC Alpha AXP architecture using AINT [Pai96], a cycle-accurate pipeline-level simulator, with a superscalar back-end. We configured the simulator to emulate a processor similar to the DEC Alpha 21264 [KMW98]. In particular, the simulated 4-way superscalar CPU has a 128-entry instruction window, a 32-entry load/store buffer, four integer and two floating point units, a 64kB 2-way set associative L1 instruction-cache, a 64kB 2-way set associative L1 data-cache, a 4MB unified direct-mapped L2 cache, a 4096-entry BTB, and a 2048-line gshare-bimodal hybrid branch predictor. The three caches have a block size of 32 bytes. The modeled latencies are shown in Table 4.1. The functional units are fully pipelined. Operating system calls are executed but not simulated. Loads can only execute when all prior store addresses are known.

This configuration represents our baseline architecture. All the speedups reported in this paper are relative to this baseline CPU, which does not contain a load value predictor.

Instruction Type	Latency
integer multiply	8-14
conditional move	2
other int and logical	1
floating point multiply	4
floating point divide	16
other floating point	4
L1 load-to-use	1
L2 load-to-use	12
Memory load-to-use	80

Table 4.1: The functional unit and memory latencies (in cycles) of our simulator. The load-to-use latencies do not include the effective address calculation, which takes another cycle.

We performed a very detailed parameter space evaluation comprising several hundred simulation runs to obtain effective configurations for the load value predictors presented in this paper.

The best performing last n value predictor under 30kB of state that we found is our *SAg Last 4 Value Predictor* with a height of 512, a history length of ten bits, four bit saturating counters for re-execute with a threshold of nine and a penalty of three, and 5-bit counters for re-fetch with a threshold of sixteen and a penalty of sixteen. The predictor uses 16kB of state for storing values plus 5kB for the confidence estimator. Unless otherwise noted, these are the parameters used with our predictor.

Since we believe that next-generation CPUs will only contain moderately sized load value predictors, our study focuses on 16kB predictors. Nevertheless, we also present results of larger and smaller predictors.

Note that the predictors presented in this paper are optimized for speedup, which implies optimizing the predictor performance at instruction commit. The interaction between the CPU and the predictor, however, takes place in the predict and then again in the update stage, possibly long before the time of commit. This discrepancy may be an issue because, for example, the accuracy with which wrong path instructions are predicted is most likely less important than the accuracy of correct path instructions. Consequently, a high overall accuracy measured at predict or update may not be representative since it makes no statement about the prediction accuracy of the instructions that are actually retired. We found the ratio of total predicted loads over committed value predicted loads to be just under 1.5, indicating that there is a significant number of predictions that most likely have little impact on the overall performance. To account for any effects this might have, we model out-of-order and wrong-path updates of the predictor accurately in our detailed pipeline-level simulator. Also, unless otherwise noted, the results presented in this paper refer to the time of instruction commit.

4.1 Benchmarks

We use the eight integer programs of the SPEC95 benchmark suite [SPEC95] for our measurements. These programs are well understood, non-synthetic, and compute-intensive, which is ideal for processor performance measurements. Despite the lack of desktop application code in the suite, it is nevertheless quite representative thereof, as Lee et al. found [LCB+98].

We use the reference input set and the more optimized peak-versions of the programs (compiled on an Alpha 21164 using DEC GEM-CC with full optimization *-O5 -ifo*). The binaries are statically linked, which enables the linker to perform additional optimizations to further reduce the number of run-time constants that are loaded during execution. These optimizations include most of the optimizations that OM [SrWa93] performs. In spite of this high optimization level and good register allocation, 22.9% of the instructions executed by the programs are loads.

Note that the few floating point load instructions contained in the binaries are also predicted and loads to the zero-registers as well as load immediate instructions are ignored.

We execute each of the benchmark programs for 300 million instructions on our simulator after having skipped over the initialization code in "fast execution" mode. This fast-forwarding is very important because the initialization part of programs is not representative of the general program behavior [ReCa98]. Table 4.2 shows the number of instructions that were skipped (in billions) and gives other relevant information about the simulated segment of each of the eight SPECint95 programs. GCC is executed for 334 million instructions and no instructions are skipped since this amounts to one complete compilation.

Information about the Simulated Segments of the SPECint95 Benchmark Suite										
	exec	percent	skipped		L1 load	L2 load	load sites that account for			
program	instrs	loads	instrs	IPC	misrate	misrate	Q100	Q99	Q90	Q50
compress	300 M	17.9%	6.0 G	1.35	24.4%	2.8%	62	56.5%	45.2%	14.5%
gcc	334 M	23.9%	0.0 G	1.51	2.4%	6.4%	34345	41.2%	15.7%	2.5%
go	300 M	24.1%	12.0 G	1.44	1.4%	15.3%	9619	40.2%	17.9%	2.7%
ijpeg	300 M	16.8%	1.0 G	1.44	1.4%	51.3%	2757	13.7%	6.7%	1.9%
li	300 M	25.5%	4.0 G	1.99	5.4%	0.6%	419	56.6%	28.6%	10.3%
m88ksim	300 M	20.7%	1.0 G	1.25	0.1%	11.2%	747	71.9%	26.6%	3.3%
perl	300 M	31.2%	1.0 G	1.57	0.0%	46.9%	1437	15.7%	11.6%	3.1%
vortex	300 M	23.6%	5.0 G	2.89	2.2%	10.2%	1973	48.6%	18.0%	2.8%
average		22.9%		1.68	4.7%	18.1%	6420	43.0%	21.3%	5.1%

Table 4.2: This table shows, from left to right, the number of simulated instructions (in millions 'M'), the percentage of instructions that are loads, the number of skipped instructions (in billions 'G'), the instructions per cycle of the baseline architecture, the L1 data-cache load miss-rate, the L2 load miss-rate, and some quantile information. The quantile columns show the number of load sites that contribute the given percentage (e.g., Q50 = 50%) of executed loads in absolute terms for Q100 and percentages thereof for the remaining quantiles.

The results shown in Table 4.2 only take into account load instructions within the 300 million simulated instructions of each of the benchmarks. However, we found the eight segments to be very representative of the complete programs, as full program executions revealed. For example, the predictability of the entire programs is within five percent of the numbers measured for the simulated segments.

Except for compress, all the programs have a quite low L1 data-cache load miss-rate. Some of the reported L2 load miss-rates are quite large. However, the corresponding number of accesses is very small and hence the large miss-rates do not have a significant impact on the performance.

An interesting point is the relatively small number of load sites that contribute most of the executed load instructions. For example, 5% of the load sites that are executed at least once account for 50% of the dynamically executed loads and only 43% of the executed load sites account for 99% of the executed loads.

4.2 Averaging Speedups

In this paper, we use the term speedup to denote how much faster our baseline processor becomes when a load value predictor is added.

Speedup results are normally obtained by executing a suite of programs on a simulated version of the enhanced CPU. To get a meaningful estimate of the expected performance improvement that a load value predictor will deliver, the speedup results of the individual benchmark programs need to be averaged. This is often done using the arithmetic or geometric mean even though neither mean is suitable for the task. For example, if a predictor reduces the runtime of one program to one half of what it used to be and does not affect the runtime of three more programs, then the expected speedup is $4/3.5 \approx 1.143$, which is the old combined normalized runtime over the new combined normalized runtime. The runtime is normalized to give every program in the suite the same weight. The expected speedup is identical to the harmonic mean of the four individual speedups (2, 1, 1, 1). Computing the arithmetic mean of the four speedups results in $5/4 = 1.25$ and the geometric mean yields $2^{(1/4)} \approx 1.189$. Both results are not only different from the harmonic mean speedup, but they are also too high and thus portray too optimistic a picture. In fact, the geometric and the arithmetic mean of any set of numbers is always greater than or equal to the harmonic mean. Therefore, all the averaged speedups presented in this paper are harmonic mean speedups.

5. Results

The following subsections describe the results. In Section 5.1 we evaluate the performance of our *Tag SAg L4V* predictor by comparing it to oracles (Section 5.1.1) and other predictors from the literature (Section 5.1.2) as well as by analyzing the operation of the individual predictor components separately (Section 5.1.3). Section 5.2 presents a sensitivity analysis of the predictor parameters. Section 5.2.1 investigates the prediction potential, Section 5.2.2 examines the predictor size versus width trade-off, Section 5.2.3 studies the history length, and Section 5.2.4 explores the parameters of the saturating counters.

To better analyze the space of parameters we only show averages over the eight benchmarks and not the individual programs. Furthermore, except for Sections 5.1.2 and 5.2.2, we restrict ourselves to predictor sizes between 16kB and 29kB since even predictors of that size perform well but are most likely not too large to be included in next generation microprocessors.

5.1 SAg L4V Predictor Overall Performance

In brief, our *Tag SAg L4V* predictor's accuracy at commit is 98.1% using re-fetch. 32.8% of the load instructions are predicted with the correct value, 0.6% with an incorrect value. This results in a harmonic mean speedup over SPECint95 of 12.5% relative to the same CPU but without the predictor. With re-execute, the accuracy of the predicted load instructions that are committed/retired is 92.9%. 36.9% of the load instructions are correctly predicted and 0.8% are incorrectly predicted. The resulting harmonic mean speedup is 13.7%.

5.1.1 Comparison with Oracles

To get a better understanding of the performance of our predictor, we modified the simulator to provide various degrees of perfect knowledge.

The first oracle (called *perf-inh*) inhibits all the incorrect predictions that the oracle-less predictor (*normal*) would make (i.e., no incorrect predictions take place).

The next oracle (*perf-ce*) incorporates a perfect confidence estimator. In addition to inhibiting all incorrect predictions, the predictor is now forced to make a prediction whenever the selected component contains the correct value.

The last oracle (*perf-ce/sel*) includes both a perfect confidence estimator and a perfect selector. This means that the oracle not only always makes a prediction if the correct value is available and never makes a prediction otherwise, but also that it chooses the component that will make a correct prediction if there is such a component. In other words, if any component in the predictor would make a correct prediction, it is selected and a prediction is made, otherwise no prediction is attempted.

Figure 5.1 shows the speedups of the oracle-less predictor and the three oracles for re-fetch and re-execute.

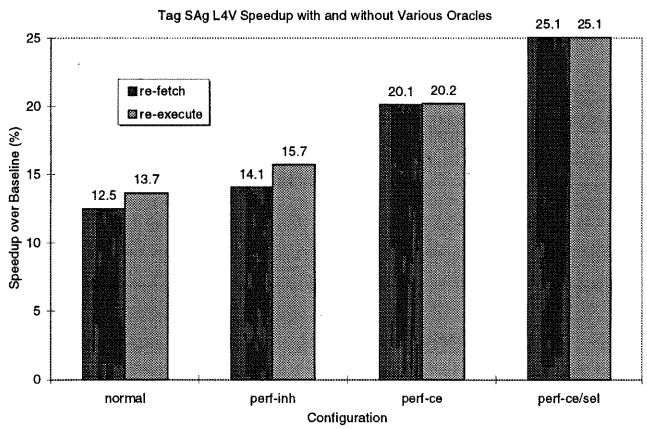


Figure 5.1: Re-fetch and re-execute speedups of various predictors with different degrees of perfect knowledge (oracles).

Inhibiting incorrect predictions (*perf-inh*) improves the speedup somewhat relative to the ordinary predictor (*normal*). The improvement is not very large, though, indicating that incorrect predictions either do not diminish the performance much or that there is only a small number of incorrect predictions to begin with. Since the predictor's accuracy is over ninety percent, the latter is the more probable explanation.

Adding perfect confidence estimation (*perf-ce*) results in a significant increase in speedup, suggesting that our imperfect CE is rather conservative. Since our CE setting is the result of a global optimization and therefore yields one of the highest speedups possible, we conclude that trading off missing potentially correct predictions for reducing the number of incorrect predictions is beneficial with the CPU we are simulating. Apparently, incorrect predictions are indeed very harmful and should therefore be avoided, meaning that a high prediction accuracy is paramount.

Note that, because there are no mispredictions, the *perf-ce* speedups for re-fetch and re-execute should be the same. The minor discrepancy in the two speedups stems from different timing behavior within the modeled CPU after a load value misprediction that affects the predictor updates, which are non-speculative. The reason for this is that *perf-ce* does not necessarily "correctly" predict wrong-path load instructions (that are executed due to branch mispredictions) since a correct load value is not always defined in such a case.

Adding a perfect confidence estimator and a perfect selector (*perf-ce/sel*) results in yet another big boost in speedup, implying that our selection mechanism could be improved. Overall, our predictor is able to reap about half of the existing potential.

5.1.2 Comparison with Other Predictors

In this section we compare several predictors from the literature with our own. The sole metric for this comparison is the harmonic mean speedup over SPECint95. To make the comparisons as fair as possible, every predictor is scaled to 16kB of state for storing values. Note that, due to the different confidence estimators, the overall predictor sizes vary between about 17kB and 29kB of state. Our predictor with 21kB is among the smaller ones.

We performed a detailed parameter space evaluation for our predictor to determine the setting that yields the highest speedup. For the predictors from the literature we use the best parameter setting indicated by the authors. Observing that the threshold has a significant effect on performance, we also varied the threshold setting of these predictors to find an optimal value. Figure 5.2 and Figure 5.3 present the resulting best mean speedup of the predictors with a re-execute and

a re-fetch misprediction recovery mechanism, respectively. For lack of a better order, the predictors are sorted by size, with the smallest being on the left side. On the far right side we show the speedup achieved by doubling the size of the 64kB L1 data cache instead of adding a load value predictor.

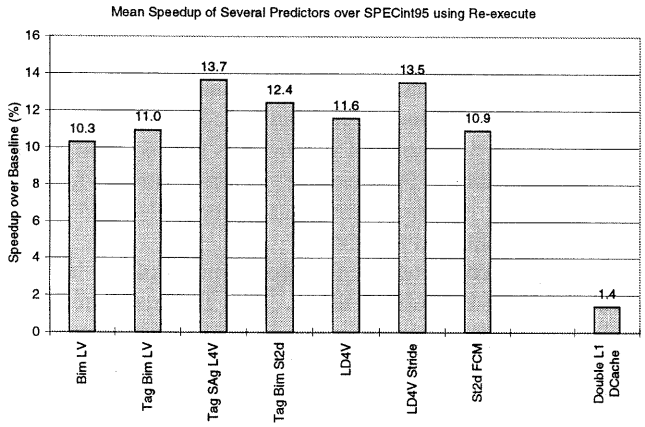


Figure 5.2: The best harmonic mean speedup of several predictors with sizes between 17kB and 27kB using a re-execute execution core.

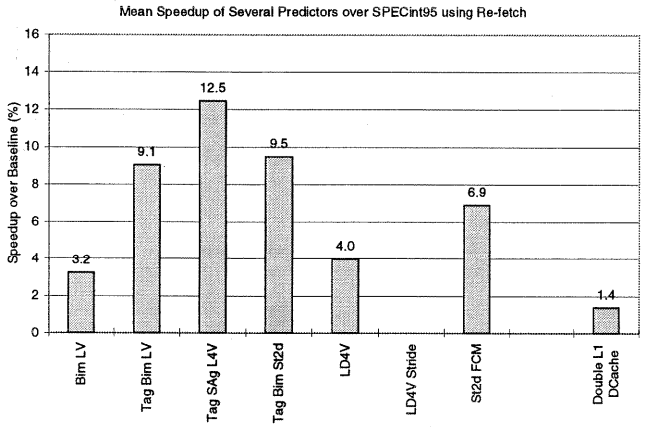


Figure 5.3: The best harmonic mean speedup of several predictors with sizes between 17kB and 29kB using a re-fetch execution core.

Our predictor (*Tag SAg L4V*) outperforms all the other predictors, including larger and significantly more complex ones. *LD4V Stride* comes close to the speedup of our predictor when re-execution is utilized, but not with re-fetch.

While all the predictors perform quite well with the more complex but more forgiving re-execution policy, their performance suffers significantly when the currently available and more realistic re-fetch misprediction recovery hardware is used. Our predictor sustains the least performance decrease. Surprisingly, its re-fetch speedup is higher than any other predictor's re-execute speedup with only one exception: *LD4V Stride* performs somewhat better with re-execute than our *Tag SAg L4V* using re-fetch. However, *LD4V Stride* performs so poorly with re-fetch that it actually slows programs down.

Note how much better *Tag Bim LV* performs than *Bim LV*. Most of the difference in speedup does not come from the partial tags but from the more adequate choice of CE parameters. In particular, changing the counter penalty from one to seven for re-fetch made the biggest difference. This significant improvement is a direct result of our detailed pipeline-level simulations.

We suspect that other proposed predictors can also be improved upon by imposing a heavier penalty on their counters. However, we do not believe that they will reach the performance of our predictor unless they switch to a SAg-based CE, since in all our measurements with otherwise identical components, bimodal predictors always turn out to be inferior.

Since our predictor performs well with re-fetch, it is possible that no re-execution core is necessary. This result is encouraging, in particular for the near future because it means that microprocessor designers can simply use the already existing branch misprediction hardware to recover from value mispredictions.

All the predictors (except *LD4V Stride* using re-fetch) outperform the doubled L1 data cache. This is surprising because doubling the cache requires over 64kB of additional state, which is almost four times as much as the predictors require when all the cache hardware is accounted for. Clearly, there is a point beyond which adding a load value predictor is likely to yield more benefit than using the same number of transistors to increase the cache size.

To get a broader perspective on the performance of the various predictors, we present Figure 5.4 and Figure 5.5, which show the speedups of the predictors for different sizes. The Figures no longer include *Bim LV* since *Tag Bim LV* outperforms it. We added *Tag SAg LV* in its place, which is identical to *Tag Bim LV* except it uses our SAg-based CE instead of the bimodal one.

As expected, with re-execute, all the predictors perform quite well across the entire range of sizes. With 16kB and above, our last four value predictor outperforms all the other predictors and the speedup gap to the second best predictor (*LD4V Stride*) increases as the predictors become larger. Note how *Tag SAg LV* consistently delivers an additional percent of speedup over *Tag Bim LV*.

For small predictor sizes, the last four value predictor is no longer tall enough to hold all the frequently executed load instructions. As a result, its performance suffers signifi-

cantly. However, for both 4kB and 8kB, *Tag SAg LV* performs about as well as the best predictors for these two sizes.

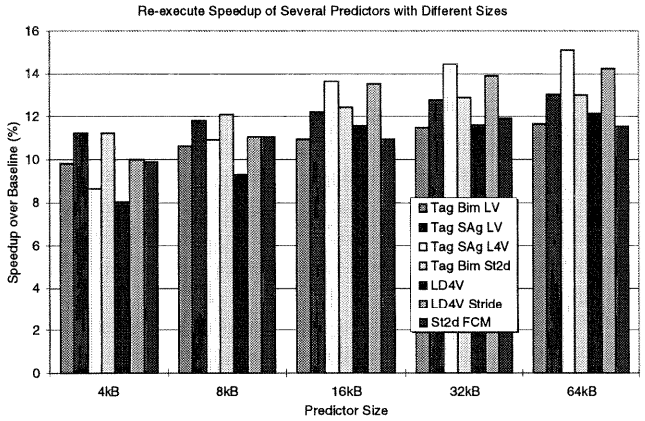


Figure 5.4: The re-execute speedup of several predictors for different sizes. The sizes refer to the amount of state used to store values and do not include the state of the confidence estimators.

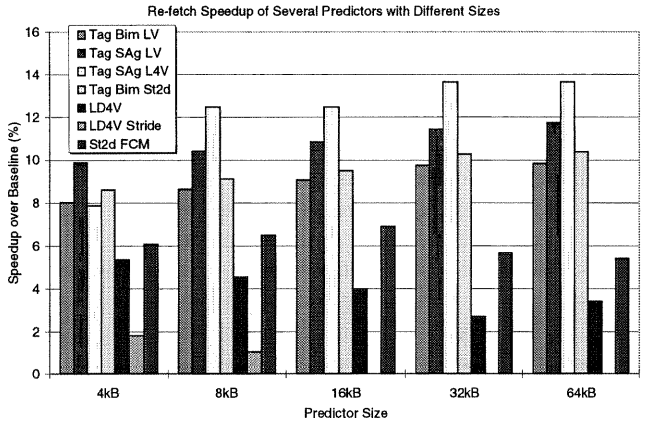


Figure 5.5: The re-fetch speedup of several predictors for different sizes. The sizes refer to the amount of state used to store values and do not include the state of the confidence estimators.

With re-fetch, the performance of most predictors is significantly lower. Our last four value predictor now outperforms the other predictors starting at a size of 8kB.

Again, *Tag SAg LV* is consistently superior to *Tag Bim LV*. We take this as strong evidence that SAg CEs are better suited for load value prediction than bimodal CEs.

For the smallest predictor size (4kB), the last four value predictor is too short and its performance is accordingly low. However, *Tag SAg LV* outperforms all the other predictors for this size. Apparently, a SAg-based CE with a last *n* value predictor (*LV* is a last one value predictor) makes a strong combination for both re-fetch and re-execute.

Note that the performance of some of the predictors from the literature actually decreases with re-fetch when increasing the predictor size.

5.1.3 Component Performance

In this section we investigate the performance of the four components of our predictor individually. To perform this survey, we had to measure the numbers at the time of prediction and not at the time of instruction commit. This means that all the wrong path load instructions are included, which are less predictable. Furthermore, we only show re-fetch results since the re-execute results are almost identical and do not provide additional insight.

Figure 5.6 compares the four components of our last four value predictor in absolute terms. The first set of four bars (*selected*) indicates how often the individual components are selected. The next set of bars (*predicted*) shows how many times the components are used to make a prediction. The third set of bars (*correct*) displays how frequently a component contributes a value that results in a correct prediction.

The remaining three sets of bars indicate the percentage of the time the four components report the maximum confidence among the four components (*maxconf*), a confidence at or above the threshold (*thresconf*), and the highest possible confidence (*topconf*), respectively. Note that, as opposed to the first three sets of bars, more than one component may be counted at the same time. As a consequence, the sum of the four bars in the first three sets cannot exceed one, whereas the sum of the four bars in the last three sets may exceed one but cannot exceed four.

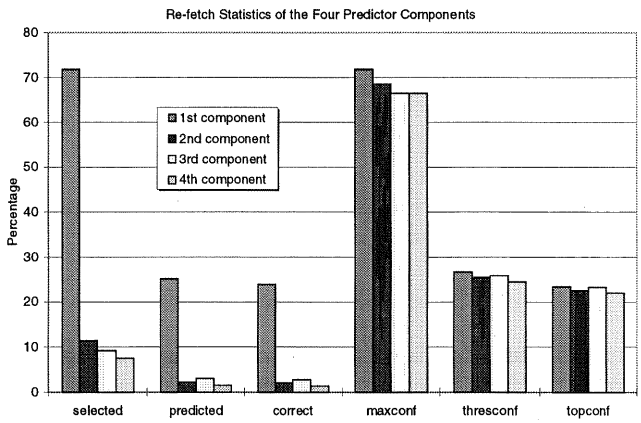


Figure 5.6: The percentage of predictor accesses that result in one of the components being selected, being used for a prediction, and causing a correct prediction as well as the percentage of time the components report the maximum confidence over the four components, a confidence above the threshold, and the highest possible confidence.

The first component, which stores the most recently loaded value, is selected much more often than the remaining components and thus performs most of the work. This is an artifact of the prioritization scheme (see Section 4).

In Figure 5.6, the bars labeled *correct* are very close to the bars labeled *predicted*, indicating that most of the attempted predictions turn out to be correct, which is in accordance with the high overall accuracy.

It is interesting to see that all the *maxconf* bars are very high. Clearly, at least one component has to report the maximum confidence at any time. However, the sum over the four bars is about 2.7, meaning that on average 2.7 components "share" the maximum confidence. We assume that this is due to counters that are saturated (i.e., multiple counters are either zero or top-1, as one would expect for highly unpredictable and highly predictable loads, respectively).

The displayed *topconf* values are close to their *thresconf* counterparts because the confidence estimator setting is very conservative, in particular with re-fetch, and only predictions with a high probability of success are made.

Interestingly, the sum over the *thresconf* bars (1.02) is much larger than the sum over the *predicted* bars (0.32), indicating that the selector eliminates 69% of the possible predictions. This does not necessarily represent a loss, however, because multiple components often make identical predictions. In fact, we found that whenever two or more components have the same maximum confidence and their confidence is not below the threshold, 99.9% of the time those components make identical predictions with re-fetch. For re-execute, the percentage is 99.6%. This is a good sign that the actual component selected in case of a tie is almost always irrelevant.

Figure 5.7 presents a relative comparison of how indicative several confidence measures are of resulting in a correct prediction. The first set of bars (*predcorr*) shows the actual accuracy of the four *Tag SAg LAV* components, that is, it shows how often an attempted prediction turned out to be correct. The second set (*maxcorr*) indicates how often the components would be correct if they made a prediction whenever they have the maximum confidence. The third set (*threscorr*) displays how often the four components would be correct if they made a prediction whenever they report a confidence at or above the threshold. The final set (*topcorr*) shows how frequently a correct prediction would be made if all the components reporting the highest possible confidence caused a prediction.

The predictor component that makes the majority of the predictions (the first component) also has the highest accuracy. This is again an artifact of the prioritization since the most predictable load values correlate highly with the times when multiple components could make a correct prediction. Consequently, whichever component is given the highest priority will make all the easy predictions and will therefore have the highest accuracy.

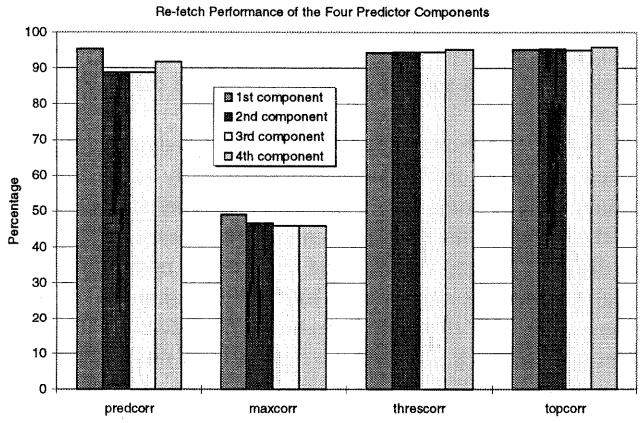


Figure 5.7: The average prediction accuracy of the four predictor components for different schemes.

Both *threscorr* and *topcorr* have higher accuracies than our predictor. This is because these schemes do not include an imperfect selector. What is interesting is that both schemes do not show considerably higher percentages than our implementable predictor. In fact, the *threscorr* accuracy of component one is roughly a percent lower than the accuracy (*predcorr*) of the same component in our predictor. This is due to the fact that in our predictor the first component is not always selected when it reports a confidence at or above the threshold, but only if it also reports the maximum confidence. This shows that the intuitive approach of selecting the most confident component does improve the prediction accuracy.

5.2 Sensitivity Analysis

5.2.1 Using Distinct Last Values

Load value predictors to exploit last value [LWS96, Gab96], stride [Gab96, SaSm97a], and finite context predictability [SaSm97a] have been studied at length in the current literature. Last *n* value predictability, on the other hand, has been less explored despite its simplicity and considerable potential. The only proposed predictor to take advantage of last *n* predictability is Wang and Franklin's last distinct four value predictor [WaFr97].

Retaining the last *n* fetched values of a given load instruction is straight forward. To make the most of the retained values, Wang and Franklin [WaFr97] suggest storing only

values that are not already stored (i.e., only distinct values). Unfortunately, this approach requires content addressable memory. Storing the last n values regardless of whether any of them are identical is much simpler. Our results in Section 5.1.2 suggest that this low complexity approach is not only more cost effective but also results in superior performance because it makes selecting one of the values easier and more accurate.

Figure 5.8 shows the prediction potential for different n when storing every loaded value versus only storing distinct values. The potential is given as the percentage of the fetched load values that are identical to at least one of the last n (distinct) fetched values. The results only take into account load instructions within the 300 million simulated instructions of each of the eight benchmarks. However, the numbers in Figure 5.8 are very representative of the generally observed predictability, as complete executions of the programs revealed.

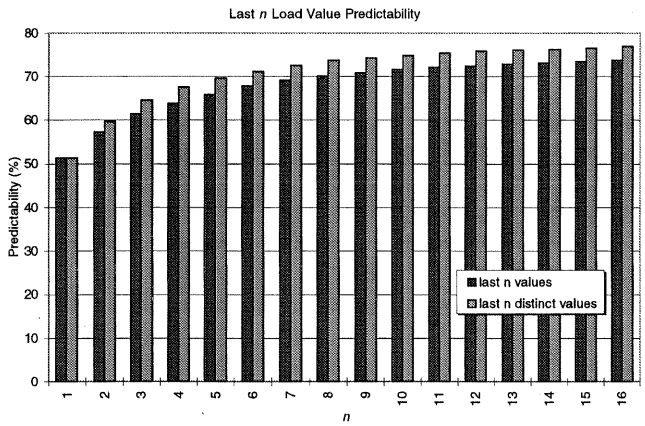


Figure 5.8: The average last n value predictability (duplicate values are allowed) and the average last n distinct value predictability (no duplicates) of the load values within the 300 million simulated instructions of each of the eight SPEC95 integer programs.

Figure 5.8 shows that larger n result in higher predictability potential. This result is intuitive since the chance of finding the correct value increases as the number of values becomes larger. The increase is considerable for small n up to about four. Then the "curve" starts flattening out and reaches saturation at approximately $n = 11$, at which point almost no extra potential is gained by further increasing n.

One very interesting observation is that for n larger than four, the potential difference between distinct and non-distinct is virtually constant (3.3%). This means that the relative advantage of storing distinct values becomes smaller as n gets larger.

For $n = 4$, which is the predictor width Wang and Franklin chose [WaFr97], the difference of 3.6% represents one eighteenth of the total potential of 64%. This means that the simpler approach of retaining not necessarily distinct values is theoretically able to perform almost as well as its more complex counterpart.

5.2.2 Predictor Size and Width

It is important that a load value predictor's height be large enough to accommodate the (load instruction) working set size (Section 4.1). If the predictor is too short, some frequently executed load instructions will have to share a predictor slot, which almost always results in detrimental aliasing. Predictors that are too tall, on the other hand, underutilize many of their slots. Consequently, once a predictor is large enough to accommodate the working set size, further increases in the predictor height will not increase the performance because the additional slots will not be utilized effectively. Instead, additional real-estate could be used to increase the amount of information stored in each slot, which should enable the predictor to make better and/or more predictions and thus improve its performance. Hence, the optimal predictor width depends on the working set size of the programs and the available real-estate for the predictor.

To better evaluate the tradeoff between predictor width and size, we present Figure 5.9 and Figure 5.10. They show the best mean speedup we were able to obtain for various predictor sizes and widths.

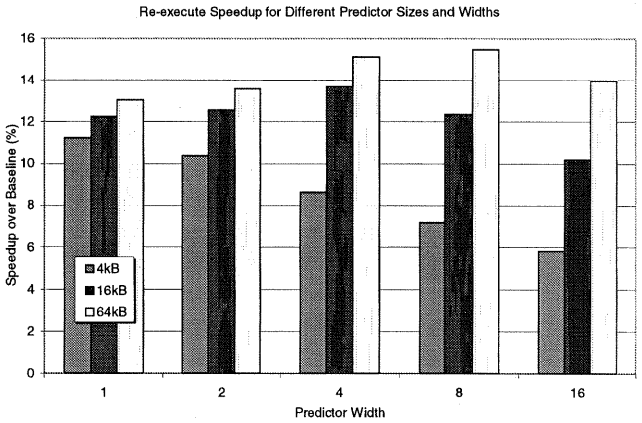


Figure 5.9: Maximum mean speedup for three predictor sizes and five predictor widths with a re-execute misprediction policy.

Figure 5.9 shows that for very small predictors (4kB of state for storing values), a width of one results in the highest

speedup. Storing two values per slot and halving the number of slots yields less speedup because there are not enough slots for the SPECint95 working set sizes, which results in more aliasing and lower performance. This effect is even more pronounced for larger widths, hence the continuous decrease in speedup as the predictor becomes wider and shorter.

With 16kB of state a width of four yields the best speedup. The detrimental aliasing only sets in above four entries per slot. When we increase the predictor size further (to 64kB), the best width turns out to be eight. Only at sixteen does the performance start to decrease again.

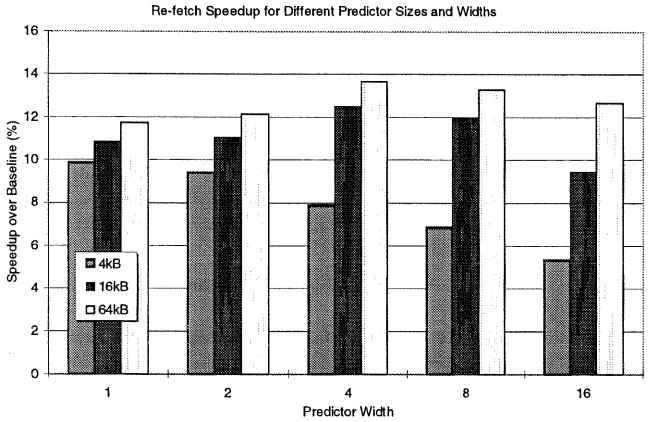


Figure 5.10: Maximum mean speedup for three predictor sizes and five predictor widths with a re-fetch misprediction policy.

Figure 5.10 is identical to Figure 5.9 except that the misprediction recovery mechanism used is re-fetch instead of re-execute. The resulting optimal predictor widths are exactly the same with one exception: a width of four (instead of eight) now yields the best speedup in the 64kB case. This change is due to the high misprediction sensibility of the re-fetch mechanism. It appears that in the 64kB case, $n = 8$ results in both more correct predictions and more incorrect predictions, which is advantageous with re-execute but harmful with re-fetch.

5.2.3 History Length

We already found history lengths of ten bits to work well for a predictor width of one [BuZo98]. Figure 5.11 shows that ten bits also mark the beginning of the saturation point for the last four value predictor, both for re-fetch and re-execute. Note that it is important not to chose the history length too large since every additional bit doubles the number of required saturating counters.

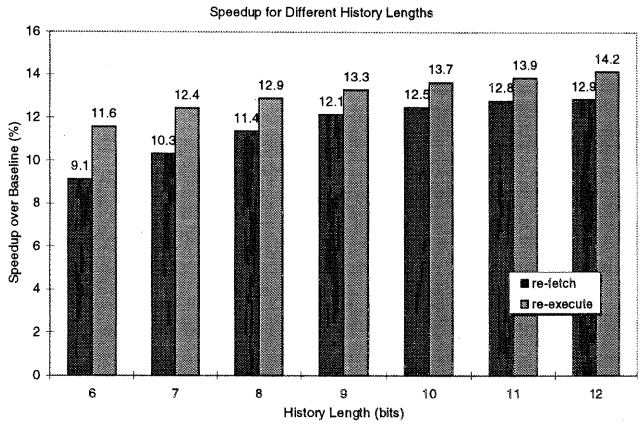


Figure 5.11: Mean speedup with respect to several history lengths.

5.2.4 Counter Parameters

For space reasons, we cannot present all the results pertaining to counter parameters but will only give a summary in this section.

With a re-execute misprediction recovery mechanism, we found four-bit saturating up/down counters to work best with our last four value predictor. Counters both smaller and larger than four bits yield considerably less performance. Consequently, we use a counter top of 16. We found a threshold of nine with a penalty of three to work quite well for this counter size, but the exact values are not very crucial. Larger thresholds with penalties of three work just as well. Likewise, a threshold of nine with a penalty above three yields about the same speedup. However, penalties of one or two result in significantly lower performance.

For re-fetch, thresholds and penalties of about half the counter top value work very well with the last four value predictor. Five, six, and seven bit counters yield the best speedup. We use the smallest of the three, which has a counter top of 32, with a threshold and a penalty of 16. Again, numbers near the ones we picked all result in approximately the same speedup.

6. Summary and Conclusions

The goal of this paper is to explore a technique to improve the accuracy and in particular the coverage of a conventional last value predictor by increasing its width to retain the last n values rather than just the last value. The only new mechanism that a last n value predictor requires is a selector that chooses which of the n values to use for the next prediction. Using the predictor's n confidence estimators for this purpose

yields good results with very little extra hardware.

Once a load value predictor is large enough to hold all the frequently executed load instructions, increasing the predictor height does not result in significantly better performance because any extra slots will not be used effectively. As an alternative, we propose using additional real-estate to increase the amount of information stored in each slot, which enables the predictor to make better and/or more predictions and thus improves its performance.

Our measurements show that, while very small predictors (4kB of state) perform the best in the conventional last value configuration, 16kB or larger predictors benefit from having a width greater than one. For example, our 16kB predictor performs best with a width of four when running SPECint95.

We use a mechanism similar to an SAg branch predictor as a confidence estimator (CE) to decide on whether to make a value prediction or not (instead of whether to predict a conditional branch to be taken or not). When comparing otherwise identical predictors, the one with our SAg CE considerably outperforms its counterparts that use other approaches.

To evaluate the performance of the predictors and to thoroughly explore the parameter space, we performed hundreds of detailed pipeline-level, cycle-accurate simulation of a superscalar high-performance microprocessor with various predictors, including several from the literature. The results show that our tagged SAg last four value predictor outperforms other predictors (that are all scaled to about the same size), including significantly more complex ones. More importantly, our predictor performs well enough with the existing re-fetch misprediction recovery mechanism that the added benefit of a more complex and not yet realized re-execution core is small in comparison.

In spite of its good performance, a comparison of our predictor with some oracles revealed that there is still significant opportunity for improvement left.

In addition to our last n value predictor, the contributions of this paper include detailed simulation results for several predictors over a wide range of parameters and sizes as well as a significantly enhanced version of a simple bimodal last value predictor.

We are currently trying to improve the predictor's performance further by hybridizing it with a stride predictor and we are investigating ways to shrink the predictor size.

Acknowledgments

This work was supported in part by the Hewlett Packard University Grants Program (including Gift No. 31041.1). We would like to especially thank Tom Christian for his sup-port of this project and Dirk Grunwald and Abhijit Paithankar for providing and helping with the pipeline-level simulator.

References

[BuZo98] M. Burtscher, B. G. Zorn. *Load Value Prediction Using Prediction Outcome Histories*. Technical Report CU-CS-873-98, University of Colorado at Boulder. October 1998.

[BuZo99] M. Burtscher, B. G. Zorn. Profile-Supported Confidence Estimation for Load Value Prediction. To appear in the *Journal of Instruction Level Parallelism (JILP)*. 1999.

[Gab96] F. Gabbay. *Speculative Execution Based on Value Prediction*. EE Department Technical Report #1080, Technion - Israel Institute of Technology. November 1996.

[KMW98] R. E. Kessler, E. J. McLellan, D. A. Webb. "The Alpha 21264 Microprocessor Architecture". *1998 International Conference on Computer Design*. October 1998.

[LCB+98] D. C. Lee, P. J. Crowley, J. J. Baer, T. E. Anderson, B. N. Bershad. "Execution Characteristics of Desktop Applications on Windows NT". *25th International Symposium on Computer Architecture*. June 1998.

[LiSh96] M. H. Lipasti, J. P. Shen. "Exceeding the Dataflow Limit via Value Prediction". *29th International Symposium on Microarchitecture*. December 1996.

[LWS96] M. H. Lipasti, C. B. Wilkerson, J. P. Shen. "Value Locality and Load Value Prediction". *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, 138-147. October 1996.

[McF93] S. McFarling. *Combining Branch Predictors*. TN 36, DEC-WRL. June 1993.

[Pai96] A. Paithankar. *AINT: A Tool for Simulation of Shared-Memory Multiprocessors*. Master's Thesis, University of Colorado at Boulder. 1996.

[ReCa98] G. Reinman, B. Calder. "Predictive Techniques for Aggressive Load Speculation". *31st Annual ACM/IEEE International Symposium on Microarchitecture*. December 1998.

[RFKS98] B. Rychlik, J. Faistl, B. Krug, J. P. Shen. "Efficacy and Performance Impact of Value Prediction". *Proceedings of the 1998 International Conference on Parallel Architectures and Compiler Technology (PACT '98)*. October 1998.

[SaSm97a] Y. Sazeides, J. E. Smith. "The Predictability of Data Values". *30th Annual ACM/IEEE International Symposium on Microarchitecture*. December 1997.

[SaSm97b] Y. Sazeides, J. E. Smith. *Implementations of Context Based Value Predictors*. Technical Report ECE-97-8, University of Wisconsin-Madison. December 1997.

[SCAP97] E. Sprangle, R. Chappell, M. Alsup, Y. Patt. "The Agree Predictor: A Mechanism for Reducing Negative Branch History Interference". *24th Annual International Symposium of Computer Architecture*, 284-291. 1997.

[SPEC95] *SPEC CPU'95*. August 1995.

[SmSo95] J. E. Smith, G. S. Sohi. "The Microarchitecture of Superscalar Processors". *Proceedings of the IEEE*. 1995.

[SrWa93] A. Srivastava, D. Wall. "A Practical System for Intermodule Code Optimization at Linktime". Journal of Programming Languages 1(1):1-18. March 1993.

[WaFr97] K. Wang, M. Franklin. "Highly Accurate Data Value Prediction using Hybrid Predictors". *30th Annual ACM/IEEE International Symposium on Microarchitecture*. December 1997.

[YePa93] T. Y. Yeh, Y. N. Patt. "A Comparison of Dynamic Branch Predictors that use Two Levels of Branch History". *20th Annual International Symposium of Computer Architecture*, 257-266. May 1993.